Regular order reductions of ordinary and delay-differential equations

J.M. Aguirregabiria [a,1], Ll. Bel [b], A. Hernández [a] and M. Rivas [a]

[a] *Física Teórica, Facultad de Ciencias, Universidad del País Vasco, Apdo. 644, 48080 Bilbao (Spain)*

[b] *Laboratoire de Gravitation et Cosmologie Relativistes, CNRS/URA 769, Université Pierre et Marie Curie, 4, place Jussieu. Tour 22-12, Boîte courrier 142, 75252 Paris Cedex 05 (France)*

Abstract

We present a C program to compute by successive approximations the regular order reduction of a large class of ordinary differential equations, which includes evolution equations in electrodynamics and gravitation. The code may also find the regular order reduction of delay-differential equations.

Key words: Ordinary differential equations; Delay-differential equations; Order reduction; Lorentz-Dirac equation; Abraham-Lorentz equation; Chaotic scattering
PACS: 02.30.Hq, 02.30.Hq, 04.50.+h, 04.90.+e, 41.60.-m, 47.52.+j
Program Library Index section: 4.3 Differential Equations

[1] Corresponding author.
e-mail: wtpagagj@lg.ehu.es
Tel.: +34 944647700 (ext. 2585)
FAX: +34 944648500

PROGRAM SUMMARY

Title of program: ODEred

Catalog identifier:

Program obtainable from: CPC Program Library, Queen's University of Belfast, N. Ireland

Licensing provisions: none

Computers: The code should work on any computer with an ANSI C compiler. It has been tested on a PC, a DEC AlphaServer 1000 and a DEC Alpha AXP 3000-800S.

Operating system: The code has been tested under Windows 95, Digital Unix v.4.08 (Rev. 564) and Open VMS V6.1.

Programming language used: ANSI C

Memory required to execute with typical data: It depends on the number of equations and retarded points stored in memory. Many intersting problems can be solved in 1 Mbyte.

No. of bytes in distributed program, including test data, etc.: 84.695

Distribution format: ASCII

Keywords: Ordinary differential equations; Delay-differential equations; Order reduction; Lorentz-Dirac equation; Abraham-Lorentz equation; Chaotic scattering

Nature of the physical problem
In different physical problems, including electrodynamics and theories of gravitation, there appear singular differential equations whose order decreases when a physical parameter takes a particular but very important value. Typically most solutions of these equations are unphysical. The regular order reduction is an equation of lower order which contains precisely the physical solutions, which are those regular in that parameter. The program computes the solution of the regular order reduction for a large set of ordinary and delay-differential equations.

Method of solution
The basic integration routine is based on the continuous Prince-Dormand method of eighth order. At each integration step, successive approximations are performed by using the polynomial interpolating the solution that has been computed in the previous approximation.

Typical running time
It depends heavily on the number and complexity of the equations and on the desired solution range. It was at most a couple of seconds in the test problems.

LONG WRITE-UP

1 Introduction

In different physical theories there appear singular evolution equations that share some common properties: most of their solutions are unphysical because their order is higher than expected except for a particular, but important, value of a parameter for which the order reduces to what one would expect on physical grounds. For instance, the Lorentz-Dirac equation [1] that describes the motion of a charged particle with radiation reaction is of third order, so that initial position and velocity would not be enough to determine its evolution and most of its mathematical solutions are 'runaway', i.e. the acceleration increases without limit. Nevertheless, in the limit when the charge goes to zero the Lorentz-Dirac equation becomes a second-order equation while the non physical solutions diverge for that value of the parameter.

Equations of this kind show strong numerical instabilities, which prevent from integrating them forward in time: even if the physical initial conditions are chosen, the integration error introduces an initially small contribution from the non physical solutions which then blow out [2,3]. The standard recipe to avoid this problem is to integrate backwards [2,3], but this is impossible in many cases because the final state from which to integrate is unknown [4].

A natural approach to this kind of problems is provided by the concept of "regular order reduction", which is an evolution equation with the right order that contains precisely the physical solutions and behaves smoothly for the particular value of the parameter for which the order of the singular equation decreases [5]. In the context of the Lorentz-Dirac equation this concept was discussed by Kerner [6] and Sanz [7]. Order reductions have been also used to replace the delay-differential equations that appear in the electrodynamics [8,9] and in non-linear optics [10], as well as to analyse fourth-order equations that appear in theories of gravitation with a quadratic Lagrangian [11] and in the study of quantum corrections to Einstein equations [12].

Except in rather trivial cases, the order reduction cannot be computed exactly and some approximation scheme is necessary. One may use a power expansion [7], but the explicit expressions become quickly too complex. Several years ago, one of us (Ll.B.) wrote a routine to find the regular order reduction of delay-differential equations. To compute numerically the order reduction of singular ordinary differential equations we have proposed and analysed in some cases a method of successive approximations [4,13].

The goal of this paper is twofold: we want to make widely available the code for

the latter method, which we developed first for a very particular system [14], and to discuss its applicability in a case that was not analysed previously: that of delay-differential equations.

2 The general problem

Let us assume that an evolution equation can be written (after inserting additional variables and trivial equations to make it a first-order system) in the form of n differential equations that in vectorial notation appear as

$$\dot{\mathbf{x}} = \mathbf{f}\left(t, \mathbf{x}, \dot{\mathbf{x}}, \ddot{\mathbf{x}}, \ldots\right), \tag{1}$$

and that we suspect on physical grounds that the successive approximations

$$\dot{\mathbf{x}} = \mathbf{f}_0\left(t, \mathbf{x}\right), \tag{2}$$

$$\vdots$$

$$\dot{\mathbf{x}} = \mathbf{f}_{n+1}\left[t, \mathbf{x}, \dot{\mathbf{g}}_n\left(t; t_0, \mathbf{x}_0\right), \ddot{\mathbf{g}}_n\left(t; t_0, \mathbf{x}_0\right), \ldots\right], \qquad (n = 0, 1, \ldots), \tag{3}$$

converge in some domain when one uses an appropriate starting point (2) that is suggested by the physical problem. In many cases, but not necessarily, all the $\mathbf{f}_n$ but the first one will have the same functional structure. The key point is that at each stage a first-order system has to be solved because higher derivatives appearing in (1) are missing in (2) and have been replaced in (3) by the (numerical) solution $\mathbf{g}_n\left(t; t_0, \mathbf{x}_0\right)$ computed in the previous approximation:

$$\dot{\mathbf{g}}_0 = \mathbf{f}_0\left(t, \mathbf{g}_0\right), \qquad \mathbf{g}_0\left(t_0; t_0, \mathbf{x}_0\right) = \mathbf{x}_0, \tag{4}$$

$$\dot{\mathbf{g}}_n = \mathbf{f}_n\left(t, \mathbf{g}_n, \dot{\mathbf{g}}_{n-1}, \ddot{\mathbf{g}}_{n-1}, \ldots\right), \qquad \mathbf{g}_n\left(t_0; t_0, \mathbf{x}_0\right) = \mathbf{x}_0. \tag{5}$$

For simplicity we have assumed above that the differential equations are ordinary, i.e. that $\mathbf{x}$ and its derivatives were evaluated at a single time t. But nothing prevents us for considering more general cases in which some values appearing on the right hand side are evaluated at one or several earlier times. In such a case we have a delay-differential system and its phase-space is infinite-dimensional because the initial condition is not a single n-dimensional point $\mathbf{x_0}$ but a full vector function $\mathbf{h}(t)$ for some appropriate interval $t_{-1} \leq t \leq t_0$ [15]. We will consider an example of this type in section 4.3.

Except in some special cases [4,13], nothing is known about the convergence of this scheme of succesive approximations and we are not claiming that the

method will converge for all (or at least a large class of) mathematical problems of this kind, but that we think that it will work for many interesting physical problems. (We refer to [9,10,4,13] for the physical motivations of our approach.) In fact one of the main reasons to write the code we will describe in the next section was that it can be used to explore numerically in which cases, and for which domains, the convergence is likely to happen. One should also keep in mind that the numerical results of the method of successive approximations can be checked *a posteriori* by solving the singular ordinary differential equation backwards from the final state computed by means of the code we describe in the next section.

3 The code

Our code attempts to construct automatically a good approximation to the result obtained in the limit, i.e. to the solution of the regular order reduction, by constructing the successive approximations (5) at each integration step.

Since the solution $\mathbf{g}_n$ appearing on the right hand side of (3) must be evaluated, along with its derivatives, for different values of t in the interval corresponding to the current integration step, we need not only the values of $\mathbf{g}_n$ on the initial and final points of the interval but a method to compute it along the whole interval. Although to test our ideas we developed first a method based on the classical fourth-order Runge-Kutta algorithm with Hermite interpolation, we present here only the final code based on the eighth-order Prince-Dormand method implemented by Hairer, Norsett and Wanner [16], as well as an alternative fifth-order code based on the Dormand-Prince method [16], which can be used for testing purposes.

Both codes are embedded Runge-Kutta methods with automatic step-size control that provide not only the values of the solution at discrete points but also a polynomial which interpolates the solution between each pair of points, allowing us to compute the desired values and derivatives. Of course, one cannot expect a good accuracy if derivatives of high order are necessary, but we have found in practice that, even in difficult chaotic cases, the seventh-order polynomial of the eighth-order method is good enough in many interesting cases where only first and second derivatives are needed. The second method provides a fourth-order interpolating polynomial which can be used if only low accuracy results are required. The availability of the interpolating polynomial is also the key feature needed to deal with delay-differential equations.

Our codes are loosely based on the routines `DOP853` and `RETARD` by Hairer, Norsett and Wanner [16]. They consist of four public functions that are declared in `ODEred.h`, which you have to include in your code to take advantage

of ANSI declarations. You have to compile and link with your code the file containing the implementation of these functions: `ODEred.c` in the case of the eighth-order method and `ODEred5.c` for the fifth-order code.

3.1 Public function `InitODEred`

Before using the main integration routine (or when one of the three parameters below changes) the following call must be made to initialise some global values used to store the solution even after the integration routine exits:

```
r = InitODEred(issyst,dimen,points);
```

3.1.1 Input parameters

`issyst` An integer that should be always equal to `TRUE` (nonzero), except when the system of n first-order equations is equivalent to a single equation of order n because the first $n-1$ equations are just in the form $\dot{x}_i = x_{i+1}$. In the latter case, `issyst` may be set to `FALSE` (0). Although an equation must be written with this package in the form of an equivalent system, this value controls the meaning of the first parameter in `InterpolateODEred` (see section 3.3), which may be used in a more natural way with this option.

`dimen` A positive integer with the number n of first-order equations. This is the dimension of the regular order reduction.

`points` A positive integer with the number of solution points to be stored in memory. In the case of ordinary differential equations 1 will be enough because the code only needs the last point to compute interpolated values and derivatives. If the equations contain delay, it is necessary to compute values for retarded values of the independent variable, so the corresponding polynomial coefficients must be still in memory. Since the code chooses automatically the step size, it is not easy to know beforehand how many points are necessary for a given accuracy, especially if the delay is not constant. In practice one can start with a value of 100 and make it higher if the `InterpolateODEred` described below returns the error code -1.

3.1.2 Return value

An integer `r` equal to -1 if the initialization failed by lack of memory, and to `points` if it was successful.

This routine and `EndODEred` could have been included inside `SolveODEred`, but we prefer this approach, because in some cases it is convenient to make

more than one call to `SolveODEred` with the same stored solution, which moreover often has to be used even after `SolveODEred` exits.

3.2 Public function `SolveODEred`

The actual numerical integration of the system is performed by using:

```
r = SolveODEred(&t,x,t0,tend,equation,output,
                eps,hmax,h,accuracy,maxiter,forget);
```

3.2.1 Input parameters

`t` Initial value for t. Passed by reference.
`x[i]` Initial values for `x`. Passed by reference.
`t0` Final t-value for initial conditions in retarded systems: the initial function `initial` (see section 3.7) will be used for $t <$ `t0`, if it is different from `NULL`. It is not automatically made equal to the initial value of t because sometimes `SolveODEred` is called more than just once with the same `t0` but different ts (for instance, when the integration is paused and then continued).
`tend` Final t-value. `tend-t` may be negative (except for delay-differential systems) to integrate backwards.
`equation` A user-provided function returning the derivatives on the right hand side of the equations in the system (see section 3.5).
`output` The output function to be called at each solution point (see section 3.6).
`eps` Local tolerance for step-size control.
`hmax` Maximum step-size.
`h` Initial step-size guess.
`accuracy` Maximum relative error among successive approximations.
`maxiter` Maximum number of iterations.
`forget` If this integer is `TRUE` previously computed solution points will be removed from memory.

3.2.2 Output parameters

`t` Last value for t. Passed by reference.
`x[i]` Final values for `x`. Passed by reference.

3.2.3 Return value

An integer equal to 0 on success, to -1 if while trying to keep the truncation error under `eps` the step size became too small, -10 if the maximum number of iterations `maxiter` was performed without attaining the desired `accuracy`. Positive values are error codes returned by `output`.

3.3 Public function `InterpolateODEred`

The user-provided functions `equation` and `output` may compute the values of the solution for any value of t by using

```
r = InterpolateODEred(n,t,x,initial);
```

As an added feature, it is also able to compute derivatives of the solution.

3.3.1 Input parameters

`n` Integer indicating the "component" to be computed. If $\texttt{n} \geq 0$, the `n`-th component will be returned. If $\texttt{n} < 0$, the components $0, \ldots, |\texttt{n}| - 1$ will be computed.
For $j = 0, \ldots, \texttt{dimen} - 1$, the j-th component is always the j-th dependent variable x_j (i.e. the j-th component of `x`).
If `issyst` is `TRUE`, the index $\texttt{n} = \texttt{dimen} \times k + j$ refer to the k-th derivative of component x_j, with $j = 0, \ldots, \texttt{dimen} - 1$ and $k = 1, 2, \ldots$
If `issyst` is `FALSE`, the system must be equivalent to a single equation of orden `dimen`, in such a way that the dependent variable x_j is the derivative of component x_{j-1} for $j = 1, \ldots, \texttt{dimen} - 1$. In this case, the index $\texttt{n} = \texttt{dimen} - 1 + k$ corresponds to the k-th derivative of component $\texttt{dimen} - 1$ and is, thus, the `n`-th derivative of the first dependent variable x_0.

`t` Independent variable for which the value(s) must be computed.

`initial` This parameter will be `NULL` in general, but can be a function defined to return from the call `initial(n,t)` with the `n`-th component of the initial function in retarded systems for values $\texttt{t} < \texttt{t0}$. See section 3.6.

3.3.2 Output parameter

`x` Pointer to a double, or, if $\texttt{n} < 0$, to an array with at least $|\texttt{n}|$ doubles, where the interpolated value(s) will be stored.

3.3.3 Return value

An integer equal to 0 on success, to -1 if `t` is out of the range stored in memory (or available through `initial`), and to -2 if `n` (or $|\mathtt{n}| - 1$, if $\mathtt{n} < 0$) is greater than the maximum value.

3.4 Public function ***EndODEred***

When the ODEred package is no longer necessary, the program must use

```
EndODEred();
```

to release the allocated memory.

On the other hand, the user must provide the following two functions when `SolveODEred` is called.

3.5 User-provided function ***equation***

When calling `SolveODEred` one has to provide a pointer to a function that, whatever its name is, has the following structure:

```
double equation(int     n,     /* Iteration number     */
                double  t,     /* Independent variable */
                double *x,     /* Dependent variables  */
                double *f)     /* Returned derivatives */
{
  ...
  f[...] = ...;
  ...
}
```

It defines the differential system and returns $\mathbf{f}_n\left(t, \mathbf{x}, \dot{\mathbf{g}}_{n-1}, \ddot{\mathbf{g}}_{n-1}, \ldots\right)$. The function knows the $\mathbf{f}_n$ it has to compute by means of the parameter `n` and, if $\mathtt{n} > 0$ uses `InterpolateODEred` to find the $\mathbf{g}_{n-1}$ values (and their derivatives), which were computed in the previous approximation. In the example of section 4.1 one could use the following function:

```
void equation(int iteration,double t,double *x,double *f)
{
  if (iteration == 0) {
    *f = -A0*(*x);
  }
```

```
  else {
    double x2;
    InterpolateODEred(2,t,&x2,NULL);
    *f = -A0*(*x)+EPSILON*x2;
  }
}
```

The first time the routine is called at each step `iteration` is null and a first-order equation is solved. In the remaining iterations the interpolating polynomial is used through the function `InterpolateODEred` to compute the value of the second derivative appearing in the right-hand-side of the complete equation.

3.6 User-provided function ***output***

When calling `SolveODEred` one has to provide a pointer to a function with the following declaration (and any name):

```
int output(int     n,  /* Iteration number     */
           double  t,  /* Independent variable */
           double *x)  /* Dependent variables  */
```

Each time an integration step has been performed `SolveODEred` calls this function with the current values of the independent and dependent variables. This allows using the computed solution (to print it, for instance). The solution and its derivatives for other values of t may be found by means of `InterpolateODEred`. The parameter `n` may be used to know how many successive approximations were necessary to attain the desired `accuracy`. The function must return a status code in a nonnegative integer: if it is different from 0, `SolveODEred` will stop immediately and return this value as an error code. This allows exiting the integration when some condition is met even before `tend` is reached (see the example in section 4.2).

3.7 User-provided function ***initial***

When dealing with delay-differential equations one may define a pointer to a function which returns the initial functions for $t <$ `t0`. The function is declared as

```
double initial(int     n,  /* Component index      */
               double  t)  /* Independent variable */
```

may have any name, and must return the value of the `n`-th component of the dependent variable `x` for $t =$ `t`. This possibility is provided for generality, so that the routine could be used to integrate delay-differential equations, but notice that in this case the package should be used with `maxiter` = `accuracy` = 0, which disables the mechanism of successive approximations. To compute the regular order reduction of a delay-differential equation one must not provide this function because the algorithm will not converge unless it happens to be the one corresponding to the as yet unknown reduction. Calling `SolveODEred` with a `NULL` pointer will instruct the algorithm to use successive approximations to find the initial functions from the initial values $\mathbf{x}_0$.

3.8 The algorithm

The algorithm is a direct implementation of (2)–(3) applied at each integration step, rather than to the whole integration range. First the next point is computed with a Prince-Dormand step by using `n` = 0 when calling `equation`, which effectively solves (2). If the integration error is estimated to be below the relative error in `eps` [16], the coefficients of the interpolating polynomial are stored and the whole step is repeated with `n` = 1, and so on until one of the following things happens:

(1) If the relative error between the last two approximations to the next solution point is below `accuracy`, the step is accepted, because this is the desired approximation to the solution of the regular order reduction in the step. This estimation of the relative error (along with the automatic step-size control) allows getting the desired approximation with the minimum computational effort. But one can also compute the same number of approximations at every step, by disabling `accuracy`, as explained now.
(2) If `accuracy` is 0 and `maxiter` iterations have been computed, the step is accepted. This is useful to compute in all steps a fixed number of approximations to explore how the algorithm works in different problems, or if one knows that further approximations cannot meaningfully improve the solution because the original equation was itself approximate [12].
(3) If `accuracy` is nonzero and `maxiter` iterations have been performed without getting a value for the relative error between approximations below `accuracy`, `SolveODEred` immediately returns with an error code equal to -10. This may happen because `accuracy` and/or `maxiter` are too low or because we are out of the domain in which the method of successive approximations converges.

In the first two cases above the coefficients in the interpolating polynomial are stored in another location, so that the information corresponding to the last `points` is available through `InterpolateODEred` to integrate delay-differential

equations and to make possible continuous output in the `output` function, which is then invoked to allow the calling program using the new solution point. If `output` does not return a positive error code and `tend` has not been reached, one continues with the next step.

If at any point the integration error is estimated to be above the relative error in `eps`, all the data corresponding to the current step is discarded and one starts again with `n` = 0 and a smaller value for the step-size computed according with the strategy discussed in [16]. In the same way, the step-size is automatically increased when the estimated truncation error is too low. If the step-size becomes too small for the working (double) precision, `SolveODEred` returns with an error code equal to -1.

If both `accuracy` and `maxiter` are null, the successive approximations are disabled and `SolveODEred` is just an ODE integrator.

4 Test cases

The code has been checked in a variety of cases by using a GUI environment designed to analyse dynamical systems [14]. Some of them have been discussed elsewhere [4,13] and we present here some new examples for which the driver programs are provided.

4.1 The simplest example

The regular order reduction of the equation

$$\dot{x} = -a_0 x + \epsilon \ddot{x}. \tag{6}$$

is

$$\dot{x} = -ax, \qquad a \equiv \frac{\sqrt{1 + 4a_0\epsilon} - 1}{2\epsilon} \tag{7}$$

and it can be seen by an argument similar to the one used in [4] that the analytical method of succesive approximations (2)–(3) will converge to it for $0 \leq a_0\epsilon < 3/4$. Since we know the solution, $x = x_0\,\mathrm{e}^{-at}$, of the regular order reduction (7) for the second-order equation (6), we can directly check the global integration error in this simple case. In the provided `TestRed1.c` file, the problem is solved for $a_0 = 1$, $\epsilon = 0.1$ and $x(0) = 1$ (a single initial condition!) with `eps` $= 10^{-10}$, `hmax` $= 1$, `accuracy` $= 10^{-8}$ and `maxiter` $= 100$.

To illustrate the result we collect in the TEST RUN OUTPUT section some solution points —which have been interpolated with `InterpolateODEred`— as well as the accumulated global error, $(x_{\rm numerical} - x_{\rm exact})/|x_{\rm exact}|$, and the number of iterations the routine needed to reach the required `accuracy`. The evolution of this error, which essentially measures the performance of the interpolating polynomial, is displayed in figure 1. When using the eighth-order method the actual integration step is rather long, about 0.45, and each interpolating polynomial is used for a wide domain, which explains the relatively large variation of the error over a single step. In the same figure we see that if one lowers the maximum step size by setting `hmax` $= 0.1$ the error behaves more smoothly and is about one order of magnitude lower. For comparison, we have included the result when the fifth order routine is used: the error is clearly higher and far more steps are needed, so that one obtains the same results with `hmax` $= 0.1$ and 1. We have found in practice that this happens in very different problems: the fifth-order method can be used only if modest accuracy is required and the eighth-order method and the associated interpolating polynomial behave rather well, especially if after some essays one guess the right value for `hmax`.

4.2 Chaotic scattering and Abraham-Lorentz equation

Of course, the code would not be useful if it were able to deal only with linear equations. To check it under far more difficult conditions we will consider the chaotic scattering of a charged particle with radiation reaction moving in a plane around four other equal particles held at rest in the vertices $\mathbf{x}_i$ of a square (see figure 2). The equation of motion is in this case the Lorentz-Dirac equation [1] and the problem has been analysed first in [4]. Here we will consider a simplified case in which we use the non-relativistic approximation given by the Abraham-Lorentz equation, which is in this case:

$$\frac{{\rm d}^2\mathbf{x}}{{\rm d}t^2} = \frac{e^2}{m}\sum_{i=1}^{4}\frac{\mathbf{x}-\mathbf{x}_i}{|\mathbf{x}-\mathbf{x}_i|^3} + \tau_0\frac{{\rm d}^3\mathbf{x}}{{\rm d}t^3}, \qquad \tau_0 \equiv \frac{2e^2}{3mc^3}. \tag{8}$$

This is a singular third-order equation, which reduces to a second-order equation both when the particle charge $e \to 0$ and when one neglects the radiation reaction ($\tau_0 \to 0$). Any of these two second-order equations can be used as starting point for the succesive approximations, because if one starts from the free case, $\ddot{\mathbf{x}} = 0$, the next approximation is just the second-order equation obtained by taking $\tau_0 = 0$ in (8). For plane motion, one has to consider four first-order equations for the derivatives of the position and velocity components. In figure 2 we see the first approximations one gets with `TestRed2.c` by setting `accuracy` $= 0$ and `maxiter` $= 0, 1, 2, 3, 4$, as well as the approximation to the regular reduction obtained with `accuracy` $= 10^{-10}$, which is labelled as

AL and needs between 4 and 19 iterations depending on the charge position and its distance to the fixed charges.

In this case we do not know the exact solution, but we can check the quality of the solution computed by our code by integrating backwards from the end point towards the starting one by means of any ordinary differential equation solver, because the unphysical modes are exponentially damped out when integrating backwards [2]. Integrating backwards with `ODEred` —which can be used as a good solver for ordinary and delay differential equations— is around ten times faster than integrating forward in this problem, but notice that trying to guess the final state from which to integrate backwards to the desired initial conditions would be impossible in a capture process in which nothing is known a priori about the final state, and very costly in the example we are analysing due to the sensitive dependence on initial conditions associated to chaos and the fact that the final velocity may be very different (around 25% lower in the trajectory displayed in figure 2) from the initial one due to the radiation. In `TestRed2.c` to perform this backward integration one simply applies the same `SolveODEred` (with `maxiter` = `accuracy` = 0 to disable the successive approximations) to the six first-order equations necessary to write the third-order system in a plane. The initial conditions now include the derivative of the acceleration, which is computed by `InterpolateODEred` from the numerical solution constructed to approximate the regular order reduction in the forward integration. The result of this backward integration is also displayed in figure 2, where it is indistinguishable from the forward integration AL in which the code constructed the order reduction. The detailed error, $|x_{\rm forward} - x_{\rm backward}|/(|x_{\rm forward}| + |x_{\rm backward}|)$, is displayed in figure 3.

4.3 A delay-differential equation

Our code is also able to deal with delay-differential equations in which some arguments appear evaluated at retarded values of the independent variable. To check the algorithm in this context we will consider the following linear but illustrative example:

$$\dot{x}(t) = -ax(t-r), \qquad \text{with } a, r > 0, \tag{9}$$

where, in fact, the only dimensionless parameter is ar. (More interesting but far more complex retarded equations appear in non-linear optics [10] and electrodynamics [8,9].) This is an infinite-dimensional system because the initial condition is a full function defined in $[t_0 - r, t_0]$ but it is singular and becomes a first-order ordinary equation both when $a \to 0$ and $r \to 0$. We are interested

in the corresponding regular order reduction, which happens to be

$$\dot{x}(t) = \frac{W(-ar)}{r} x(t), \tag{10}$$

where W is the principal branch of the Lambert W function, which is defined implicitly in $W\,\mathrm{e}^W = z$ as a generalization of the logarithm [17]. Furthermore, by using the fact that $z^{z^{z^{\cdot^{\cdot^{\cdot}}}}} = -W(-\ln z)/\ln z$ [17], it can be seen that the succesive approximations (2)–(3) will converge to (10) for $0 \leq ar < \mathrm{e}^{-1}$. One can check this by using `TestRed3.c`, as shown in the TEST RUN OUTPUT and in figure 4, for $a = 1$, $r = 0.3$, $x(0) = 1$. We want to stress that since at each approximation an ordinary differential function has to be solved, no `initial` function has to be provided but only the initial value.

Acknowledgements

The work of J.M.A., A.H. and M.R. has been partially supported by The University of the Basque Country under contract UPV/EHU 172.310-EB036/95. J.M.A. and M.R have also been supported by DGICYT project PB96-0250.

References

[1] F. Rohrlich, Classical Charged Particles, (Addison-Wesley, Reading, MA, 1965).

[2] J. Huschilt and W.E. Baylis, Phys. Rev. D 13 (1976) 3256.

[3] W.E. Baylis and J. Huschilt, Phys. Rev. D 13 (1976) 3262.

[4] J.M. Aguirregabiria, J. Phys. A 30 (1997) 2391.

[5] H.J. Bhabha, Phys. Rev. 70 (1946) 759.

[6] E. Kerner, J. Math. Phys. 6 (1965) 1218.

[7] J.L. Sanz, J. Math. Phys. 20 (1979) 2334.

[8] Ll. Bel and X. Fustero, Ann. Inst. H. Poincaré 25 (1976) 411; and references therein.

[9] Ll. Bel, in: Relativistic Action at a Distance: Classical and Quantum Aspects, ed. J. Llosa (Springer, Berlin, 1982), p. 21.

[10] Ll. Bel, J.-L. Boulanger and N. Deruelle, Phys. Rev. A 37 (1988) 1563.

[11] Ll. Bel and H. Sirousse-Zia, Phys. Rev. D 32 (1985) 3128.

[12] L. Parker and J.Z. Simon, Phys. Rev. D 47 (1993) 1339.

[13] J.M. Aguirregabiria, A. Hernández and M. Rivas, J. Phys. A 30 (1997) L651–L654.

[14] J.M. Aguirregabiria, DS Workbench beta version. This unpublished program is a greatly improved version for Windows 3.1, 95 and NT 4.0 of: ODE Workbench, Physics Academic Software, (American Institute of Physics, New York, 1994).

[15] R.E. Bellman and K.L. Cooke, Differential-Difference Equations (Academic Press, New York, 1963).

[16] E. Hairer, S.P. Norsett and G. Wanner, Solving Ordinary Differential Equations I: Nonstiff Problems, 2nd. ed. (Springer, Berlin, 1993).

[17] R.M. Corless, G.H. Gonnet, D.E.G. Hare, D.J. Jeffrey and D.E. Knuth, On the Lambert W Function, Technical Report CS-93-03, Dept. Comp. Sci., University of Waterloo (1993).
`ftp://cs-archive.uwaterloo.ca/cs-archive/CS-93-03/W.ps.Z`.

TEST RUN OUTPUT

Output from ***TestRed1.c***

```
t = 1 x = 0.400084  Error = -9.65801e-07 Iterations = 15
t = 2 x = 0.160067  Error = -2.36572e-06 Iterations = 15
t = 3 x = 0.0640403 Error = -4.06969e-06 Iterations = 15
t = 4 x = 0.0256215 Error = -6.01646e-06 Iterations = 15
t = 5 x = 0.0102507 Error = -8.41758e-06 Iterations = 14
```

Output from ***TestRed3.c***

```
t = 0.1      x = 0.849485    Error = 7.31759e-05 Iterations = 46
t = 0.397345 x = 0.523003    Error = 4.58862e-05 Iterations = 48
t = 0.704736 x = 0.316763    Error = 4.47788e-05 Iterations = 48
t = 1.02387  x = 0.188211    Error = 4.35142e-05 Iterations = 48
t = 1.35798  x = 0.109132    Error = 4.19486e-05 Iterations = 48
t = 1.7106   x = 0.0613964   Error = 4.00809e-05 Iterations = 48
t = 2.08563  x = 0.0333008   Error = 3.79189e-05 Iterations = 48
t = 2.48743  x = 0.0172902   Error = 3.54728e-05 Iterations = 48
t = 2.92103  x = 0.00852357  Error = 3.27512e-05 Iterations = 48
t = 3.39243  x = 0.00395061  Error = 2.97599e-05 Iterations = 48
t = 3.90905  x = 0.00170085  Error = 2.65055e-05 Iterations = 48
t = 4.48039  x = 0.000669746 Error = 2.30013e-05 Iterations = 48
t = 5        x = 0.000286944 Error = 2.63023e-05 Iterations = 48
```

FIGURE CAPTIONS

Figure 1. Evolution of the accumulated error when computing by means `TestRed1.c` the regular order reduction (7) of (6).

Figure 2. Solutions of (8) corresponding to `maxiter` $= 0, 1, 2, 3, 4$ with `accuracy` $=$ 0. The solution of the regular order reduction of (8) and the one obtained integrating it backwards are indistinguishable and appear labelled as AL.

Figure 3. Evolution of the relative distance between the solutions obtained integrating forward and backward system (8) by means of `TestRed2.c`.

Figure 4. Evolution of the accumulated error when computing by means `TestRed3.c` the regular order reduction (10) of (9).

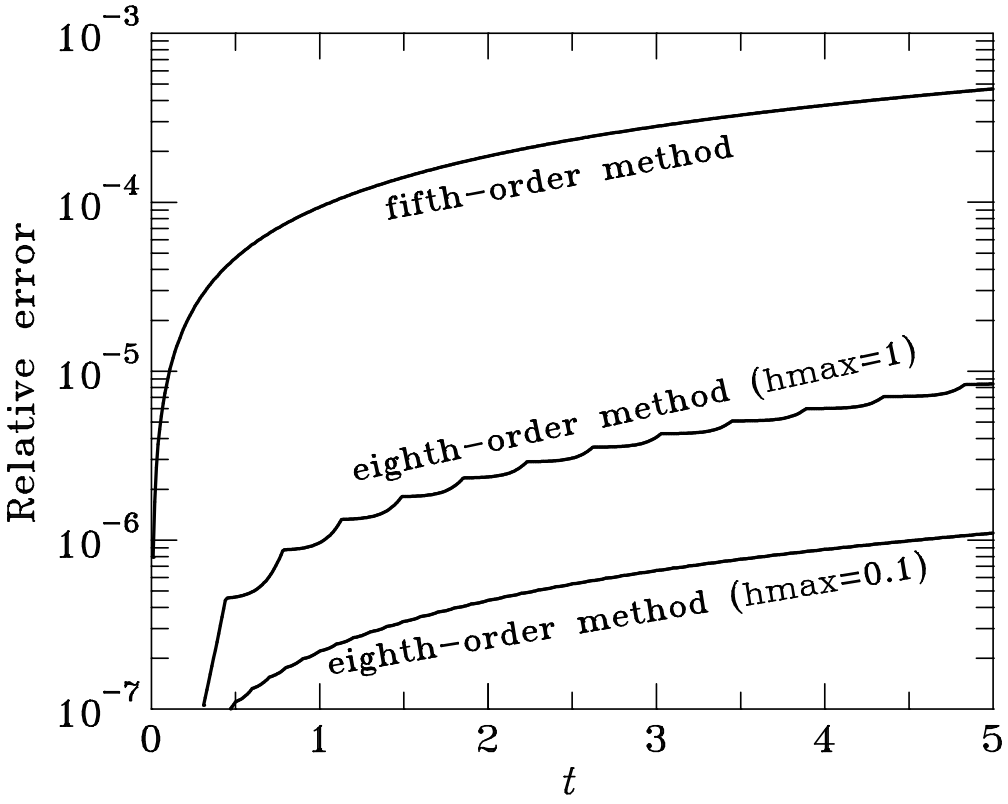

Figure 1. Evolution of the accumulated error when computing by means `TestRed1.c` the regular order reduction (7) of (6).

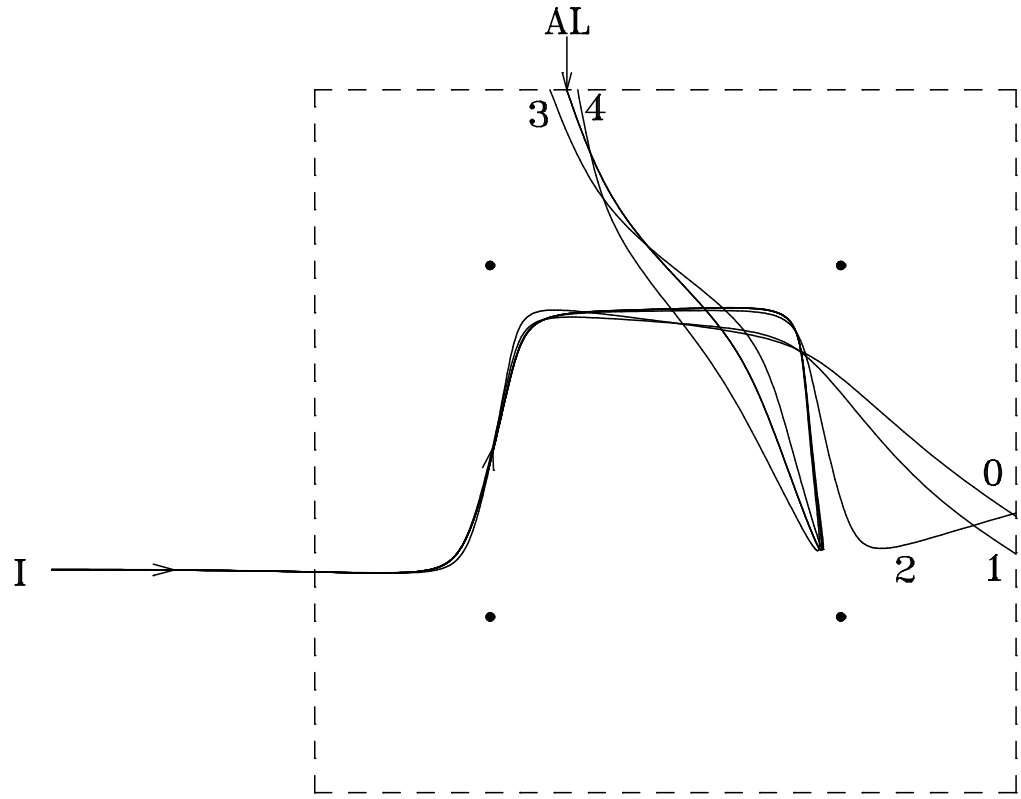

Figure 2. Solutions of (8) corresponding to `maxiter` $= 0, 1, 2, 3, 4$ with `accuracy` $=$ 0. The solution of the regular order reduction of (8) and the one obtained integrating it backwards are indistinguishable and appear labelled as AL.

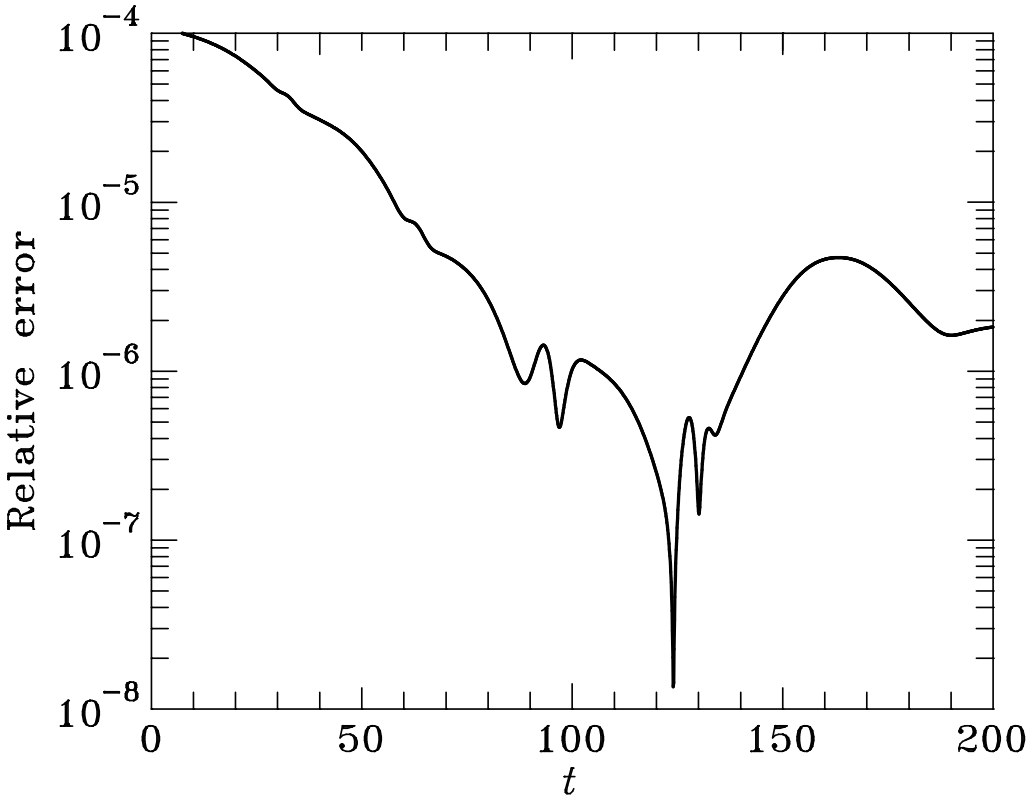

Figure 3. Evolution of the relative distance between the solutions obtained integrating forward and backward system (8) by means of `TestRed2.c`.

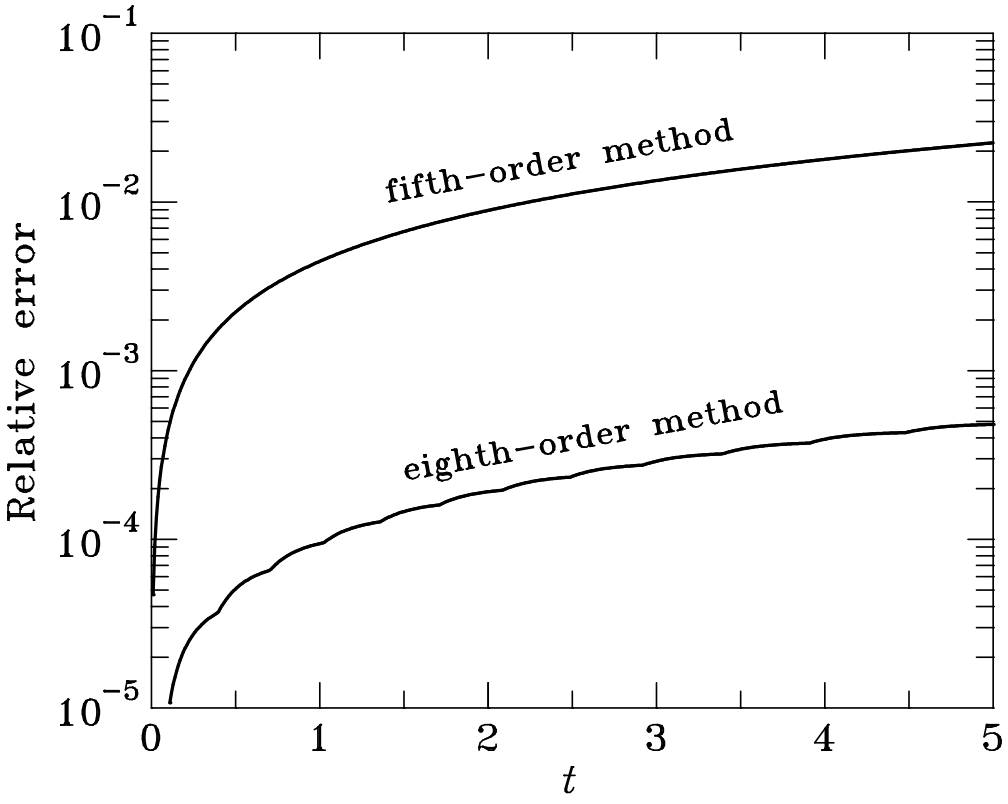

Figure 4. Evolution of the accumulated error when computing by means `TestRed3.c` the regular order reduction (10) of (9).